Lokalizacja i mapowanie w środowisku symulacyjnym MORSE

Dokumentacja projektu

Autorzy:
Bieliński Bartosz
Ciborowski Tomasz
Florczak Łukasz
Kowalski Adrian
Marocka Sabina
Martek Katarzyna
Nyka Adam
Rutkowski Wojciech
Smusz Krzysztof

I. Opis projektu.

Głównym zagadnieniem implementowanym przez nasz zespół w tym projekcie był SLAM (*ang. Simultaneous localization and mapping*), czyli estymowanie przebytej przez robota trajektorii przy jednoczesnym mapowaniu otoczenia. Algorytmów rozwiązujących ten problem jest wiele, jednak wszystkie z nich opierają się na okresowym pobieraniu danych z czujników i określaniu na ich podstawie położenia mijanych przeszkód oraz robota. Problem jednoczesnego mapowania i lokalizacji nie został jeszcze ostatecznie rozwiązany i wszystkie rozwiązania są obarczone błędami wynikającymi głównie z szumów (dodanych), które powstają przy generowaniu danych oraz niedokładności pomiarowych.

SLAM zrealizowany w ramach naszego projektu został zaimplementowany w środowisku symulacyjnym MORSE. Udało nam się osiągnąć zadowalające efekty mapowania i estymacji trajektorii w przestrzeni trójwymiarowej. W tym celu wykorzystaliśmy czujniki mające swoje odpowiedniki w rzeczywistym świecie. W programie Blender zostały stworzone mapy imitujące prawdziwe otoczenie potrzebne do przeprowadzenia symulacji.

II. Narzędzia wykorzystywane w projekcie:

- Python 3.6
- MORSE 1.3
- Blender 2.7
- Linux Ubuntu 18.04

III. Problemy i rozwiązania:

A. Model robota:

Wykorzystywanym przez nas modelem robota jest iRobot ATRV platform. Jest to prosty robot mobilny z możliwością zamontowania na nim szerokiej puli czujników. Robot poruszany jest przez silnik MotionVW (Linear and angular speed (V, W) actuator).
Wykorzystywane przez nas czujniki to:

- Video Camera
- Depth Camera
- Lidar 2D
- IMU
- Enkoder położenia (Differential odometry)

B. Otoczenia robota:

Stworzone przez nas otoczenia robota dzielą się ze względu na wielkość, ilość przeszkód, które staną naszemu robotowi na drodze, nierówności terenu oraz skomplikowanie tekstur pokrywających widoczne elementy. W ten sposób stworzyliśmy kilka rodzajów map:

- ❖ Mapy płaskie:
  - ➢ „Ściany”:
    - ■ Nasza najprostsza mapa. Zaprojektowana na rzucie zbliżonym do kwadratu, z jedną ze ścian wygiętą w łuk. Nie posiada żadnych przeszkód poza ścianami. Istnieje w wersji z kolorowymi teksturami i bez nich. Używana w celu testowania autonomicznego poruszania się wzdłuż ściany.

  - ➢ „Tunel”:
    - ■ Druga z map, które stworzone były na początku. Zaprojektowana na rzucie prostokąta, z jednym wymiarem 4 razy dłuższym od drugiego. Na środku mapy, między zakładanym miejscem startowym i końcowym robota, ustawione zostały dwie przeszkody. Ta mapa również posiada dwie wersje kolorystyczne tekstur. Mapa ta potrzebna nam była, aby zaimplementować i przetestować algorytmy omijania przeszkód

- ❖ Mapy 3D:
  - ➢ „Rampa”:
    - ■ Mała mapka, składająca się z dwóch poziomów połączonych rampą o równym nachyleniu. Stworzyliśmy ją w wersji z kolorami, jak i bez, a także sprawdziliśmy różnice jakie wprowadza dodanie sufitu. Dzięki niej przetestować

mogliśmy naszego robota w środowisku, po którym poruszać się musi w więcej niż jednej płaszczyźnie.

- ➢ „Rampa XL”:
  - Rozszerzenie początkowej mapy z rampą. Mapa ta jest kilkukrotnie większa i posiada wiele przeszkód, jednak jej podłożem wciąż są dwa poziomy połączone rampą o równym nachyleniu. Przy jej wykonaniu użyliśmy unikalnych tekstur obrazkowych zamiast zwykłych kolorów rozróżniających przeszkody. Mapa posiada wersje z sufitem oraz bez.

- ➢ „Mapa ostateczna”:
  - Duża mapa łącząca wszystkie zagadnienia wcześniej implementowane w naszych mapach, rozwinięte o dodatkowe utrudnienia. Mapa posiada różnice w poziomach w postaci pojedynczych górek i grzbietów. Występują w niej przeszkody o unikalnych teksturach. Wersje tej mapy dzielą się na posiadającą sufit, bez sufitu oraz z przezroczystym sufitem.

C. Sterowanie:

W celu przejechania jak największej powierzchni mapy i zmapowaniu jej, zostało zaimplementowany algorytm sterowania. Dzięki znajomości swojej trajektorii oraz wykorzystując dane z lidaru robot jest w stanie omijać przeszkody. Dodatkowo, znając przebytą trajektorię, robot z każdym kolejnym krokiem zawęża obszar poszukiwań w celu dokładniejszego zmapowania jak największej powierzchni terenu.

D. Komunikacja pomiędzy skryptami:

Jako *middleware* do przekazywania danych z symulacji do skryptów zdecydowaliśmy się na wbudowane w symulator MORSE sockety. Dane z socketów są przekazywane jako obiekt binarny, który po zdekodowaniu *utf-8* daje nam obiekt JSON, który można następnie zdeserializować aby otrzymać obiekt Pythonowy. Sockety do przesyłania danych wykorzystują localhost, dzięki czemu możliwe jest zdefiniowanie portu dla każdego czujnika.

Wymiana informacji pomiędzy pracującymi równolegle skryptami odpowiedzialnymi za pobieranie aktualnych danych z czujników odbywa się poprzez wywołanie odpowiedniej metody z klasy odpowiadającej danemu

czujnikowi. Obiekty tych klas działają na wątkach i opierają się na pythonowych socketach, po stworzeniu, obiekt klasy nawiązuje połączenie z odpowiednim socketem, na który kierowane są aktualne dane z czujnika. Obiekty tych klas następnie przetwarzają odpowiednio te dane i umożliwiają pobranie najbardziej aktualnych danych z czujnika. Adresy na które kierowane są poszczególne ruchy można znaleźć w pliku *env.py*.

Należy także pamiętać, żeby dać obiektom tych klas czas na pobranie i zdekodowanie pierwszych danych przed próbą ich pobrania.

Dostępne klasy i ich metody:

- **Reader w dataStreamReader.py**:
  Jest to klasa, która odbiera dane z IMU w formacie bitowym, następnie dokonuje konwersji bity-string, string-python dictionary.

  Metody:
  - *init(host, port)* - Inicjator klasy, parametrami wejściowymi są:
    - host - adres IPv4 komputera z którym łączy się obiekt klasy, np. *'localhost'*
    - port - port, z którym obiekt klasy łączy się z użyciem socketów.

  - *get(arg)* - zwraca konkretne pole z python dictionary z danymi z IMU, pole to jest wybierane poprzez parametr *'arg'*, np. *get('angular_velocity')*.

- **DepthCameraServer w depth_camera_server.py:**
  Jest to klasa, która odbiera dane z Depth Camera w formacie bitowym, następnie dokonuje konwersji bity-string, string-python dictionary.
  Punkty z Depth Camera znajdują się pod polem 'points' i po otrzymaniu są w formacie base64, z którego klasa na końcu je dekoduje i przechowuje w celu udostępnienia w zmiennej *'current_3D_points'*.

  Metody:
  - *init(host, port)* - Inicjator klasy, parametrami wejściowymi są:
    - host - adres IPv4 komputera, np. *'localhost'*
    - port - port, z którym obiekt klasy łączy się z użyciem socketów.

  - *get_all()* - zwraca cały python dictionary z takimi polami jak: *'timestamp'*, *'height'*, *'width'* oraz *'points'*.

  - *get_points()* - zwraca zdekodowane z base64 punktu z kamery głębi.

- **VideoCameraServer w video_camera_server.py:**
  Jest to klasa, która odbiera dane z Video Camera w formacie bitowym, następnie dokonuje konwersji bity-string, string-python dictionary.
  Obraz z Video Camera jest to zdjęcie RGBA, znajduje się pod polem *'image'* w python dictionary i po otrzymaniu jest w formacie base64, z którego klasa na końcu go dekoduje do postaci binarnej i przechowuje w celu udostępniania w zmiennej *'current_image'*.

  Metody:
  - *init(host, port)* - Inicjator klasy, parametrami wejściowymi są:
    - host - adres IPv4 komputera, np. *'localhost'*
    - port - port, z którym obiekt klasy łączy się z użyciem socketów.

  - *get_all()* - zwraca cały python dictionary z takimi polami jak: *'timestamp'*, *'height'*, *'width'* oraz *'image'*.

  - *get_image()* - zwraca zdekodowane z base64 obraz RGBA z Video Camera.

- **Lidar_server w sensors_classes.py:**
  Jest to klasa, która odbiera dane z Lidaru w formacie bitowym, następnie dokonuje konwersji bity-string, string-python dictionary.

  Metody:
  - *init(host, port)* - Inicjator klasy, parametrami wejściowymi są:
    - host - adres IPv4 komputera z którym łączy się obiekt klasy, np. *'localhost'*
    - port - port, z którym obiekt klasy łączy się z użyciem socketów.

  - *get_all()* - zwraca cały python dictionary z takimi polami jak: *'timestamp'* oraz *'point_list'*.

  - *get(arg)* - zwraca konkretne pole z python dictionary z danymi z lidaru, pole to jest wybierane poprzez parametr *'arg'*, np. *get('point_list')*.

- **Pose_server w sensors_classes.py:**
  Jest to klasa, która odbiera dane z Pose w formacie bitowym, następnie dokonuje konwersji bity-string, string-python dictionary.

Pose jest niewykorzystywany przez nas w procesie wyznaczania trajektorii, ale służył do testowania skuteczności naszych algorytmów. Zwraca on trajektorię najbardziej zbliżoną do idealnej.

Metody:

- *init(host, port)* - Inicjator klasy, parametrami wejściowymi są:
  - host - adres IPv4 komputera z którym łączy się obiekt klasy, np. *'localhost'*
  - port - port, z którym obiekt klasy łączy się z użyciem socketów.

- *get_all()* - zwraca cały python dictionary z takimi polami jak: *'timestamp', 'x', 'y', 'z', 'yaw', 'pitch'* oraz *'roll'*.

- *get(arg)* - zwraca konkretne pole z python dictionary z danymi z Pose, pole to jest wybierane poprzez parametr *'arg'*, np. *get('x')*.

Możliwe jest także zapisanie danych do plików csv. Uniwersalna metoda zapisu danych została przedstawiona w skrypcie *'save_data.py'*.

E. Sterowanie:

Skrypt lukasz_control.py zapewnia sterowania, gdzie robot zatrzymuje się przed każdą zmianą kierunku przy czym również wystawia flagę, że tak się dzieje. Każdy ruch wykonuje się przez wywołanie metody update().

F. Estymacja trajektorii:

1. Enkoder położenia

Morse oferuje trzy rodzaje enkoderów: integrated odometry (zwracana jest globalna pozycja), differential odometry (zwracana jest różnica pozycji między aktualnym krokiem a poprzednim), raw odometry (zwracana jest odległość krzywoliniowa od ostatniego tiku). Zdecydowaliśmy się na zastosowanie czujnika differential odometry. Na początku ustalana jest początkowa pozycja robota i dodawana jest do listy pozycji. Następnie pobierane są wszystkie dane z czujnika (używając odpowiedniego serwera) i aktualizowana jest pozycja robota. Dodatkowo należało znormalizować kąty yaw, pitch, roll.

Zaimplementowano klasę Odometry, której metoda *get_data* zwraca aktualną, globalną pozycję robota.
Wykorzystywane klasy (bardziej szczegółowe opisy klas i ich parametrów wejściowych oraz wyjściowych) znajdują się w docstring-ach w odpowiednich skryptach.

2. Odometria wizyjna

Kolejnym algorytmem do wyznaczania trajektorii robota jest algorytm Visual Odometry. Wyznacza on Punkty Charakterystyczne i ich Deskryptory na kolejnych klatkach z Video Camera, z użyciem Deskryptorów znajduje pary odpowiadających sobie Punktów Charakterystycznych pomiędzy kolejnymi klatkami i na podstawie ich przemieszczeń i odpowiednich parametrów kamery wylicza rzeczywiste przemieszczenie robota.

Wykorzystywane klasy (bardziej szczegółowe opisy klas i ich parametrów wejściowych oraz wyjściowych) znajdują się w docstring-ach w odpowiednich skryptach:

- **ImagesProcessing w images_processing_class.py:**
  Jest to klasa oferująca szereg metod umożliwiających tworzenie różnych typów obrazów z danych odbieranych z Video Camera, a także zapis tych obrazów i ich wyświetlanie.
  Video Camera wysyła domyślnie obrazy RGBA, więc potrzebne jest odpowiednie ich przetworzenie w celu uzyskania obrazów RGB, Grayscale 1 kanałowy albo Grayscale 3 kanałowy.

  Obiekt klasy posiada zmienną *'internal_image'*, w której w razie potrzeby można przechować obraz.

  Metody:
  - *create_rgba_image(binary_image, img_height, img_width)* - Zwraca obraz RGBA stworzony z obrazu w postaci binarnej.

  - *create_rgb_image(binary_image, img_height, img_width)* - Zwraca obraz RGB stworzony z obrazu w postaci binarnej.

  - *create_gray_image(binary_image, img_height, img_width)* - Zwraca obraz Grayscale 1 kanałowy stworzony z obrazu w postaci binarnej.

- *create_gray_image_3channels(binary_image, img_height, img_width)* - Zwraca obraz Grayscale 3 kanałowy stworzony z obrazu w postaci binarnej.

- *plot_rgb_image(image)* - Wyświetla obraz RGB.

- *plot_gray_image(image)* - Wyświetla obraz Grayscale

- *save_rgb_image_timestamp(image, path)* - Zapisuje dany obraz RGB w podanym folderze umieszczając w nazwie aktualny ‘timestamp’.

- *save_rgb_image_absolute_path(imahe, path)* - Zapisuje dany obraz RGB pod podaną ścieżką i z nazwą podaną przez użytkownika.

- *save_grayscale_image_timestamp(image, path)* - Zapisuje dany obraz Grayscale w podanym folderze umieszczając w nazwie aktualny ‘timestamp’.

- *save_grayscale_image_absolute_path(imahe, path)* - Zapisuje dany obraz Grayscale pod podaną ścieżką i z nazwą podaną przez użytkownika.

- *create_color_vector*(rgb_image, how_many_3d_points) - Ze zdjęcia RGB tworzy wektor kolorów o długości odpowiadającej ilości punktów pobranej z Depth Camera w danej iteracji, ten wektor może być następnie podany jako parametr *‘facecolors’* w funkcji *‘scatter’* z biblioteki *‘matplotlib’* w celu pokolorowania punktów 3D bazując na zdjęciu RGB z tej samej iteracji.

- **FindAndMatchLandmarks w landmarks_class.py:**
  Jest to klasa oferująca wiele metod ułatwiających pracę z Punktami Charakterystycznymi. Możliwe jest stworzenie w obiekcie tej klasy wybranego Detektora oraz Matchera, a następnie wykorzystywanie ich do swoich potrzeb. Detektor znajduje się pod zmienną *‘detector’*, a Matcher pod zmienną *‘matcher’.*

  Metody:

- *create_orb(maximum_amount_of_landmarks=2000, detector_edge_threshold=31, wta_param=2)* - Tworzy w obiekcie klasy detektor ORB o wybranych parametrach.

- *create_akaze()* - Tworzy w obiekcie klasy detektor AKAZE o domyślnych parametrach.

- *create_flann_for_orb(my_table_number=6,my_key_size=12, my_multi_probe_level=1, how_many_checks=100)* - Tworzy w obiekcie klasy matcher FLANN wykorzystujący algorytm działający najlepiej z detektorem ORB.

- *landmarks_from_path_to_image(path)* - Zwraca Punkty Charakterystyczne, Deskryptory oraz obraz pobrany ze ścieżki podanej jako parametr wejściowy.

- *landmarks_from_image(img)* - Zwraca Punkty Charakterystyczne oraz Deskryptory z danego obrazu.

- *draw_markers_and_save(new_path, img, kp)* - Pod podaną ścieżką zapisuje obraz z narysowanymi na nim Punktami Charakterystycznymi.

- *draw_markers_and_show(img, kp)* - Wyświetla dany obraz z narysowanymi na nim Punktami Charakterystycznymi.

- *landmarks_from_movie(source)* - Wyświetla dany film rysując na nim w czasie rzeczywistym wykrywane Punkty Charakterystyczne.

- **PinholeCamera w vo_classes.py:**
  Jest to klasa przechowująca parametry kamery po wydobyciu ich z *'intrinsic camera matrix'*.
  Metody:
  - init(width, height, int_matrix) - Inicjator klasy, parametrami wejściowymi są:
    - width - szerokość obrazu z kamery.
    - height - wysokość obrazu z kamery.
    - int_matrix - *'intrinsic camera matrix'.*

- **VisualOdometry w vo_classes.py:**

Jest to klasa realizująca algorytm Visual Odometry. Użytkownik wywołuje jedynie metodę *'update(img)'* podając jej kolejne, następujące po sobie, klatki obrazu z np. czujnika Video Camera. Obiekt tej klasy następnie sam wylicza przybliżone położenie kamery względem punktu początkowego i udostępnia je użytkownikowi w zmiennej *'cur_coords'*.

Wszystkie parametry takie jak Punkty Charakterystyczne i Deskryptory z danego oraz poprzedniego obrazu, translacje, rotacje, koordynaty oraz parametry kamery są zapisywane w obiekcie klasy. Obiekt tej klasy tworzy także w sobie Detektor i Matcher, domyślnym Detektorem jest ORB, a domyślnym Matcherem jest FLANN z algorytmem dopasowanym do ORB.

Metody:

- *init(cam)* - Inicjator klasy, parametrem wejściowym jest obiekt klasy '*PinholeCamera'*, z której sczytywane są odpowiednie parametry kamery.

- *update(img)* - Przekazuje daną klatkę obrazu do odpowiedniej funkcji w zależności od tego, która jest to klatka. Klatka pierwsza jest przekazywana do metody *'processFirstFrame'*, a każda kolejna do metody *'processFrame'*.

- *processFirstFrame()* - Wyznacza z danego obrazu Punkty Charakterystyczne oraz Deskryptory i zapisuje je wewnątrz obiektu klasy jako punkt odniesienia dla Matcherów.

- *processFrame()* - Na podstawie Punktów Charakterystycznych i Deskryptorów z aktualnego oraz poprzedniego obrazu, z użyciem algorytmu Visual Odometry, wyznacza aktualne położenie kamery względem punktu początkowego.

Zostały stworzone także skrypty umożliwiające przetestowanie poszczególnych algorytmów, szczegóły działania są opisane w samych skryptach.

Algorytm wyznaczania trajektorii z użyciem Visual Odometry w czasie rzeczywistym może być przetestowany w skrypcie

‘VO_live_test.py’. Do porównania brana jest trajektoria z czujnika Pose.

Możliwe jest także przetestowanie Visual Odometry na zapisanych, postępujących po sobie, klatkach z kamery, odbywa się to w skrypcie *‘VO_offline_test.py’*. Kolejne klatki z kamery są wczytywane ze wskazanego folderu. Wymagane klatki można zapisać z użyciem funkcji ‘*main’* w skrypcie ‘*images_processing_class.py’.*

Należy także pamiętać, że trajektoria z Visual Odometry musi zostać odpowiednio przeskalowana aby być zbliżona do rzeczywistej. Współczynnik skali może być dostosowany na początku skryptów testowych przez zmienną *‘scale’*. Wartość tego współczynnika może zostać wyznaczona eksperymentalnie lub wyliczona z prędkości robota i szybkości wykonywania się potrzebnych obliczeń.

Dodatkowo można przeprowadzić testy działania różnych Detektorów i Matcherów w skrypcie *‘feature_matchers_test.py’*. Skrypt pobiera wszystkie zdjęcia ze wskazanego folderu, następnie bierze po dwa zdjęcia, wykrywa na nich Punkty Charakterystyczne, łączy te odpowiadające sobie i zapisuje pod wskazaną ścieżką obraz składający się z dwóch, następujących po sobie klatek wraz z połączonymi pomiędzy nimi odpowiadającymi sobie Punktami Charakterystycznymi.

Możliwe do przetestowania detektory:

- ORB z parametrem WTA_K = 2
- ORB z parametrem WTA_K = 4
- AKAZE

Możliwe do przetestowania matchery:

- BFMatcher
- BFMatcher wykorzystujący algorytm *knnMatch*
- FlannBasedMatcher z parametrami dopasowanymi do detektora ORB
- FlannBasedMatcher z parametrami dopasowanymi do detektorów SURF i SIFT.
- FlannBasedMatcher wyszukujący podany na zdjęciu obiekt na innym zdjęciu.

Wybór detektora i matchera odbywa się poprzez odpowiednie ustawienie flag na początku skryptu

Możliwe jest także przeprowadzanie testów algorytmu *Optical Flow* w skrypcie *'testing_optical_flow.py'*.
Zaimplementowane testy opisane są w dokumentacji wewnątrz skryptu.

3. Lidar + IMU

Jest to kolejny sposób, w którym wyznaczana jest trajektoria robota. Brany jest w niej kąt wyliczany przez algorytm Madgwicka (opis niżej) na podstawie danych z IMU oraz różnica od poprzedniej próbki z Lidaru. Następnie z prostych zależności trygonometrycznych wyliczane jest przesunięcie w każdym kierunku. W metodzie wyliczającej to jest również pobierana flaga z algorytmu sterowania mówiąca o tym czy robot zmienił ruch np. z ruchu w przód na skręcanie. Jeśli tak się dzieje to ustalany jest nowa odległość referencyjna, od której będzie mierzona delta od poprzedniego ruchu.

Madgwick:
Został zaimplementowany algorytm, który bierze dane z sensorów zawartych w IMU, tj. akcelerometru, żyroskopu i magnetometru i dokonuje ich FUZJI uzyskując dość dokładną orientację.
Wykorzystywane w nim są kwaterniony, których klasa również została specjalnie napisana. Algorytm ten działa na osobnym wątku, gdzie pobiera najbardziej aktualne dane z czujników dzięki czemu on sam również zawsze daje najbardziej aktualną orientację.

4. Filtr Kalmana

Filtr dokonujący estymacji pozycji porównujący trajektorię przewidywaną z uzyskaną na podstawie czego określa wzmocnienie.

G. Mapowanie:

1. Mapowanie przy pomocy kamery głębi

Czujnikiem, wykorzystywanym do mapowania terenu podczas jazdy robota była Depth Camera, której użycie umożliwiło pobieranie chmury punktów w formacie base64. Po ich odkodowaniu otrzymane punkty (w formacie FLOAT) można było rozmieścić na globalnym wykresie po dokonaniu odpowiednich rotacji oraz translacji. Aby mapowanie było jak najbardziej dokładne została zmniejszona rozdzielczość kamery oraz zakres jej widzenia z domyślnych wartości na ustalone eksperymentalnie. Pobieranie danych z kamery głębokości odbywa się poprzez wywołanie odpowiedniej metody z klasy DepthCameraServer. Sam czujnik został obrócony o mały kąt (pi/8), aby zapobiec wyrysowywaniu się podłoża, po którym porusza się iRobot (ten sam zabieg został zastosowany przy Video Camera). Zdekodowane punkty zostały zaokrąglone do jednego miejsca po przecinku w celu uproszczenia obliczeń oraz zmniejszenia ich ilości. Tuż przed wyrysowywaniem gotowej mapy skrypt jeszcze raz przegląda całą listę punktów sczytanych z DepthCamery i usuwa duplikaty. Do pobierania aktualnego położenia robota (do wykonania odpowiednich przekształceń otrzymanego obrazu) wykorzystywana jest klasa Odometry.

W przypadku mapowania w czasie rzeczywistym punkty 3D są także kolorowane na podstawie zdjęcia RGB pobranego z użyciem Video Camera w tej samej iteracji co aktualnie dodawane do wykresu *'scatter'* punkty 3D, odbywa się to poprzez podanie odpowiedniego wektora kolorów na parametr *'facecolors'* w funkcji *'scatter'*. Wektor ten można utworzyć z użyciem metody *'create_color_vector'* w klasie *DepthCamera*. Jest to metoda identyczna z tak samo nazwaną metodą w klasie *ImagesProcessing*. W przypadku braku możliwości wyciągnięcia kolorów ze zdjęcia RGB, punkty 3D są kolorowane na szaro.

Wykorzystywane klasy (bardziej szczegółowe opisy klas i ich parametrów wejściowych oraz wyjściowych) znajdują się w docstring-ach w odpowiednich skryptach.

2. Fuzja danych lidaru i kamery głębi

Zakładamy, że dane z lidaru są dokładniejsze niż dane z kamery głębi ze względu na fakt, że kamera głębi jest obarczona szumami oraz niedokładnością pomiarową wynikającą z właściwości samej kamery. W trakcie mapowania za pomocą kamery głębi pobieramy również dane z lidaru w postaci słownika. Z niego wykorzystujemy współrzędne do

przeszkód względem robota. Poddajemy je translacji i rotacji i w połączeniu z położeniem otrzymanym z odometrii otrzymujemy globalne współrzędne przeszkód. Następnie przeszukujemy listy z współrzędnymi pozyskanymi z kamery głębi i porównujemy je ze współrzędnymi pozyskanymi z lidaru w celu znalezienia podobnych punktów z ustaloną wcześniej dokładnością. Na podstawie różnicy otrzymanej po porównaniu tych punktów obliczamy korektę jaką należy zastosować dla punktów z kamery głębi. Następnie przesuwamy punkty z kamery głębi o określony współczynnik korekcji.

H. Graficzny Interfejs użytkownika:

-

GUI w projekcie powstało z wykorzystaniem biblioteki PyQt5. Od strony użytkownika widzimy następujące jego elementy: okno zawierające dwie mapy 2D, dwie mapy 3D, przycisk służący do uruchomienia Morse, pole pobierające od użytkownika czas w którym ma odbywać się symulacja, przycisk rozpoczęcia i zatrzymania symulacji. Plik GUI.py zawiera następujące elementy:

- klasę MorseGUI - główne okno programu, w którym zmieniane są sceny
- klasę LogoWidget - odpowiada za wyświetlenie logo projektu po uruchomieniu GUI
- klasę MainGUI - prezentuje wyniki symulacji oraz odpowiada za interakcję z użytkownikiem. Zawiera następujące metody:
    - initUI - tworzy elementy GUI, do których nie będzie wymagany późniejszy dostęp
    - blender_callback, start_callback, stop_callback - obsługa przycisków, w kolejności: uruchamianie Morse, rozpoczęcie pracy naszych skryptów, przerwanie pracy naszych skryptów oraz wyłączenie Blender'a wraz z Morse
    - background_tasks - obsługa wątku z naszymi skryptami

I. Plik instalacyjny:

Korzystając z komend bash, napisaliśmy skrypt, który po ściągnięciu projektu z GitLab, zainstaluje wszystkie niezbędne rzeczy do jego funkcjonowania. Początkowo sprawdzamy czy użytkownik posiada na swoim systemie pythona, jeśli nie jest on instalowany. Następnie sprawdzamy czy użytkownik posiada PIP, później instalujemy Pipenv, tworzymy wirtualne środowisko, aby umieścić w nim wymagane biblioteki. Następnie sprawdzamy obecność Blender'a i jego zgodność

z Morse Robot Simulation. Finalnie przechodzimy do instalacji samego środowiska Morse.

Aby rozpocząć instalację środowiska należy otworzyć w terminalu folder zawierający plik *install.sh.* następnie wpisać następujące dwa polecenia:
chmod +x ./install.sh

./install.sh